



SYSTEM CALL IMPLEMENTATION

CS124 – Operating Systems
Winter 2015-2016, Lecture 14

User Processes and System Calls

- Previously stated that user applications interact with the kernel via system calls
- Typically invoked via a trap instruction
 - An intentional software-generated exception
- The kernel registers a handler for a specific trap
 - `int $0x80` for Linux system calls
 - `int $0x2e` for Windows system calls
 - `int $0x30` for Pintos system calls
- Can't easily pass arguments to system calls on the stack
 - Trap instruction causes the CPU to switch operating modes (i.e. from user mode to kernel mode)
 - Different operating modes have different stacks

User Processes and System Calls (2)

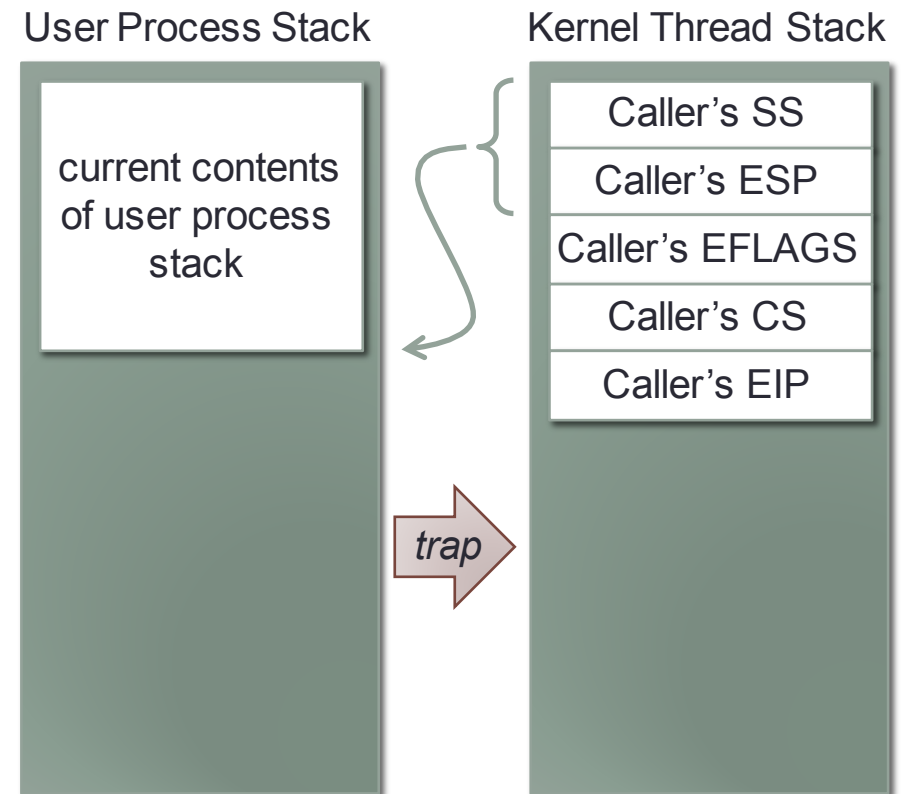
- Typically, arguments to system calls passed in registers, and the return-value(s) come back in registers
- One of the arguments is an integer indicating which system call to invoke
 - e.g. on Linux and Windows, `%eax` is set to operation to perform
 - e.g. on UNIX systems, `sys/syscall.h` specifies these numbers
 - Note: UNIX syscall IDs are not uniform across different UNIXes
- Obvious constraint: system-call arguments can't be wider than the registers
- Several possible approaches:
 - Can split larger arguments across multiple registers
 - Can store larger arguments in a struct, then pass a pointer to the struct as an argument

User Processes and System Calls (3)

- The operating system frequently exposes system calls via a standard library
 - e.g. UNIX syscalls are exposed via the C standard library (`libc`)
 - e.g. Windows syscalls are exposed via the (largely undocumented) Native API (`ntapi.dll`)
- The library serves as an intermediary between apps and the operating system
- Some functions are direct wrappers for system calls
 - e.g. `ssize_t read(int fd, void *buf, size_t nbyte)`
 - Implementation stores arguments from stack into registers, invokes the system call entry-point (e.g. `int $0x80`), and returns result
- Others utilize system call wrappers internally
 - e.g. `malloc()` is mainly implemented in user space, but uses system calls to increase the process' heap size

Review: Interrupt Mechanics

- Previously discussed how interrupts and traps are handled on IA32 (see lecture 9 for details)
 - User process has its own stack
 - Executing the trap causes the CPU to switch to the kernel-mode stack associated with the process
- Since system calls change from user mode to kernel mode, IA32 saves pointer to previous stack on new stack
- Next, CPU saves the user process' execution state: **cs**, **eip** and **eflags**



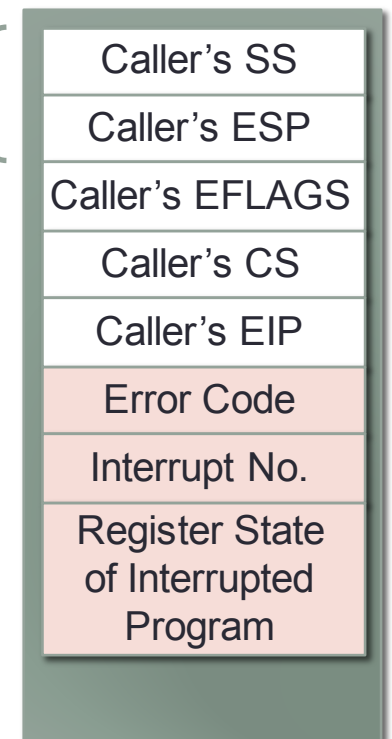
Review: Interrupt Mechanics (2)

- Operating system has a stub for every possible interrupt
- Some interrupts push an error code onto the stack; if not, the OS stub will push a dummy value for consistency
- Next, stub pushes the interrupt number onto the stack
- Finally, stub records all register state onto kernel stack
- Now the ISR can run without disrupting the interrupted code

User Process Stack

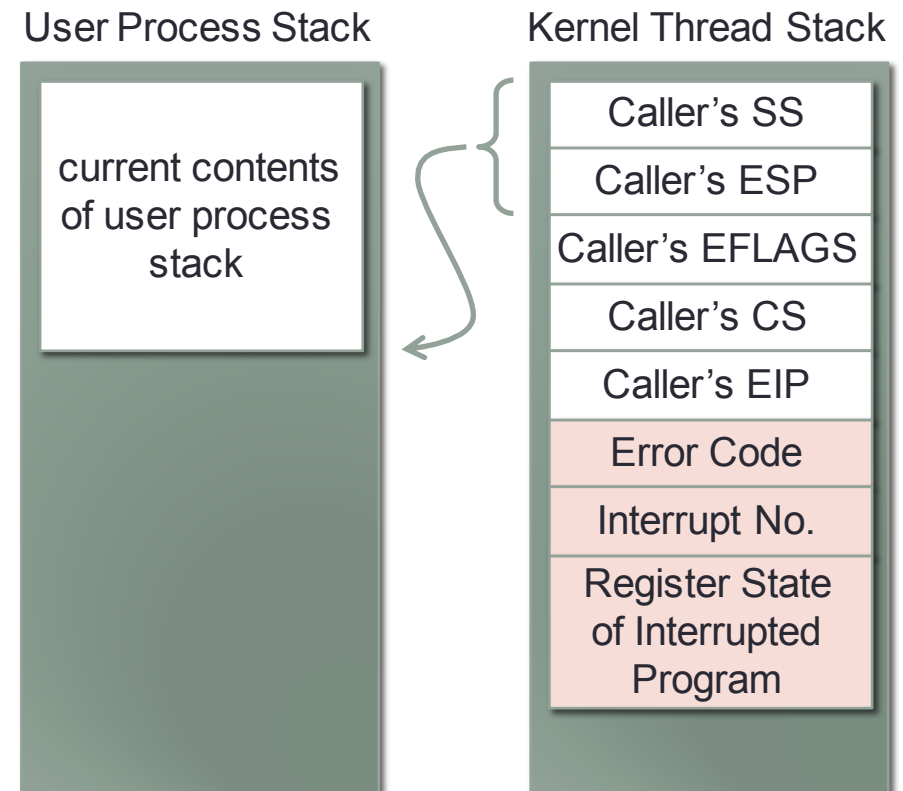


Kernel Thread Stack



System Call Mechanics

- The OS exposes the user program's CPU and register state as arguments to the ISR
 - Typically exposed to ISR as a struct with a field for each register
- System call handler needs to receive arguments from the user program
 - Can easily access these values on the kernel stack
- Syscall handler must also return a status result in **eax**
 - Can modify user program's **eax** on the kernel stack
 - When kernel returns to the user program, its context is restored
 - Program sees new value of **eax**



System Call Mechanics (2)

- The ID of the system call is used to dispatch to a function that implements the system call
 - Called a **system call service routine**
- System call service routines are usually named after their user-mode entry points
 - e.g. `sys_write()` implements `write()`
 - e.g. `sys_fork()` implements `fork()`
 - (Aside: these service routines are sometimes called within the kernel implementation to implement more complex operations)
- A system call table holds an array of function pointers to all system call service routines
 - The syscall ID is used to index into this table when making the call

System Call Mechanics (3)

- Need to check the system call ID to ensure it's valid...
 - If it's invalid, return **ENOSYS** “Function not implemented” error
- Can easily check that the ID is below the max syscall ID
- If a specific syscall ID below the max is not supported, simply register a service routine that returns **ENOSYS**

Example: Linux System Calls

- Snippet [paraphrased] of Linux `system_call()` handler:

```
...    # Save registers onto stack

# Make sure it's a valid syscall ID
cmpl $(NR_syscalls), %eax
jb nobadsys

# Return-value of syscall() will be in eax
# as usual, so set value of eax stored on
# kernel stack to ENOSYS to indicate error
movl $(-ENOSYS), 24(%esp)
jmp ret_from_sys_call
nobadsys:
...
```

Example: Linux System Calls (2)

- Linux `system_call()` handler, continued:

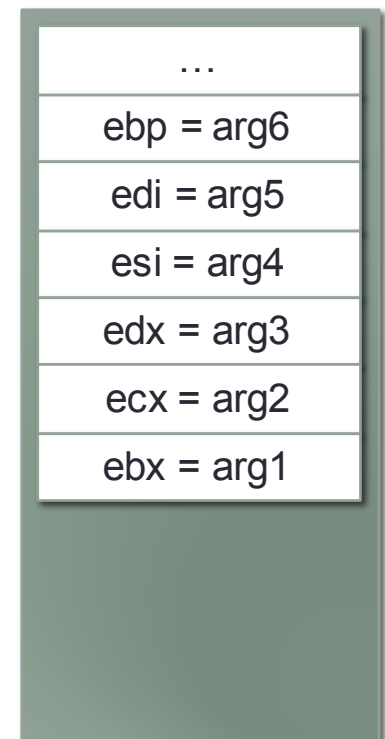
```
...
nobadsys:
    # Dispatch to the function in the system-call
    # table corresponding to the specified ID
    # (On IA32, pointers are 4 bytes, so use
    # ID*4 as the address within the table)
    call *sys_call_table(, %eax, 4)

    # Store return-value from routine into
    # location of eax on the kernel stack
    movl %eax, 24(%esp)
    jmp ret_from_sys_call
```

Example: Linux System Calls (3)

- Different syscalls require different numbers of arguments
 - e.g. `getpid()` and `fork()` require no arguments
 - e.g. `mmap()` requires up to six arguments
- System-call arguments are passed from the user process in specific registers
 - **ebx** is first argument, **ecx** is second argument, etc.
- Syscall service routines are written in C, and they expect their args on the kernel stack
- Linux `system_call()` handler pushes all of the process' registers onto the kernel stack in a specific order
 - Specifically, the reverse order that registers are used to pass arguments to system calls

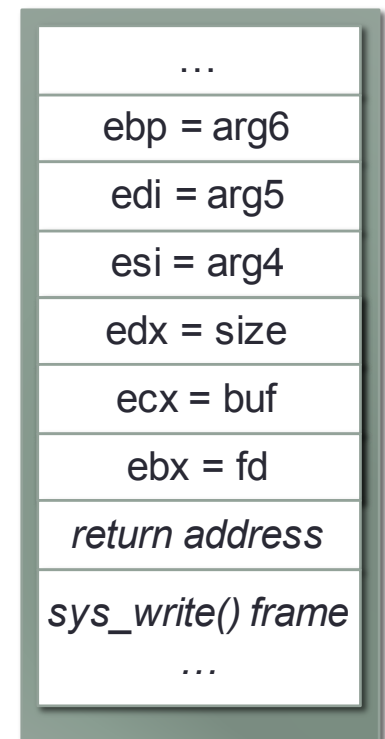
Kernel Thread Stack



Example: Linux System Calls (4)

- Arguments to syscall service routines are pushed in reverse order, following the cdecl calling convention
- Under cdecl, if a function is passed more arguments than it expects, the extra arguments are ignored
- Allows **system_call()** to dispatch to all the different service routines, regardless of the number of arguments they take
- e.g. `int sys_write(int fd, char *buf, int size)`
 - Service routine for `write(int fd, char *buf, int size)`
- When `system_call()` dispatches to `sys_write()`, `sys_write()` sees only the expected arguments
 - Extra arguments are simply ignored by `sys_write()`

Kernel Thread Stack



System Calls: Security Holes?

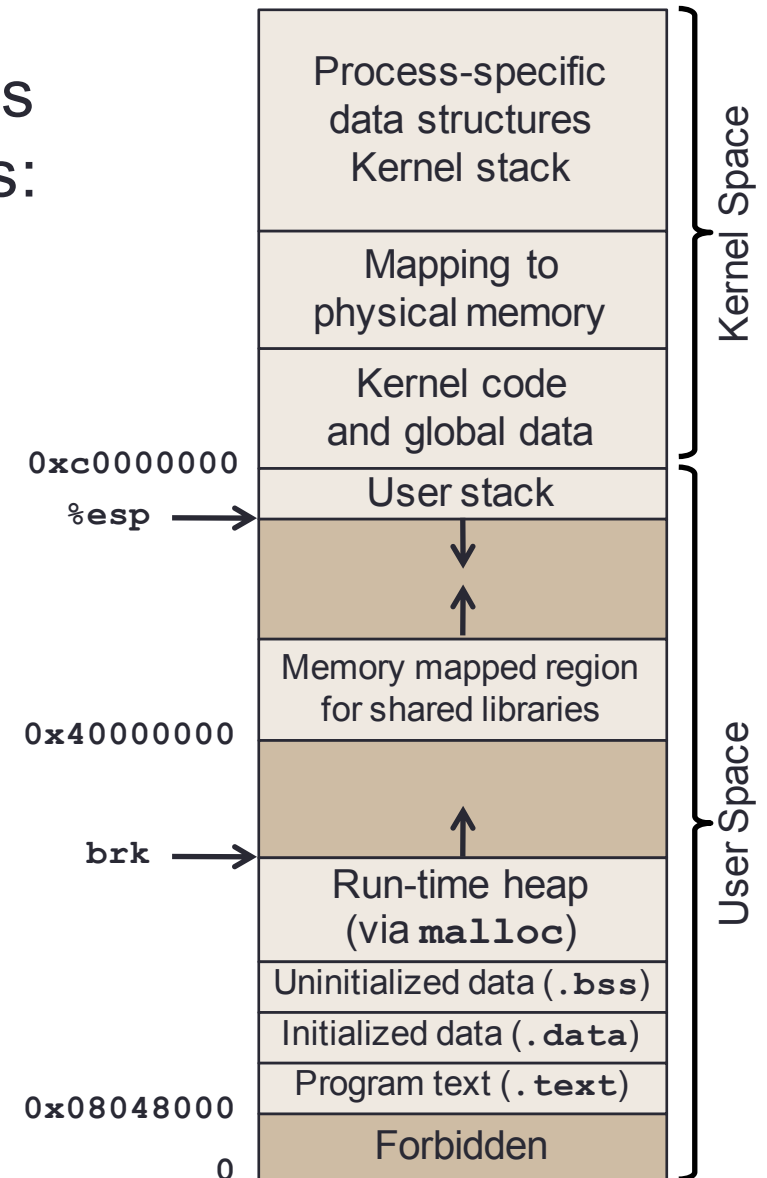
- It goes without saying that the system call service routine must carefully check all arguments to the system call...
- Are there potential security holes in accepting pointers as arguments to system calls?
- Example: `ssize_t read(int fd, void *buf, size_t nbytes)`
 - Reads bytes from a file descriptor into a buffer
- Caller specifies:
 - The file-descriptor to read
 - A pointer to the buffer to store the data in
 - A number of bytes to read

System Calls: Security Holes?!

- Example: `ssize_t read(int fd, void *buf, size_t nbytes)`
- Generally the pointers are expected to be in user space...
- What if the user-mode program specifies an address in the kernel's address space?
 - As long as the user-mode program doesn't access this address, it won't cause a general protection fault...
- But, the kernel is allowed to write to this address!
 - If kernel naively accepts the address from the user-mode program, it could overwrite critical data
- Example: target critical kernel data structures
 - Program opens file containing the data it wants to insert into kernel
 - Program passes that file descriptor and address of kernel struct...

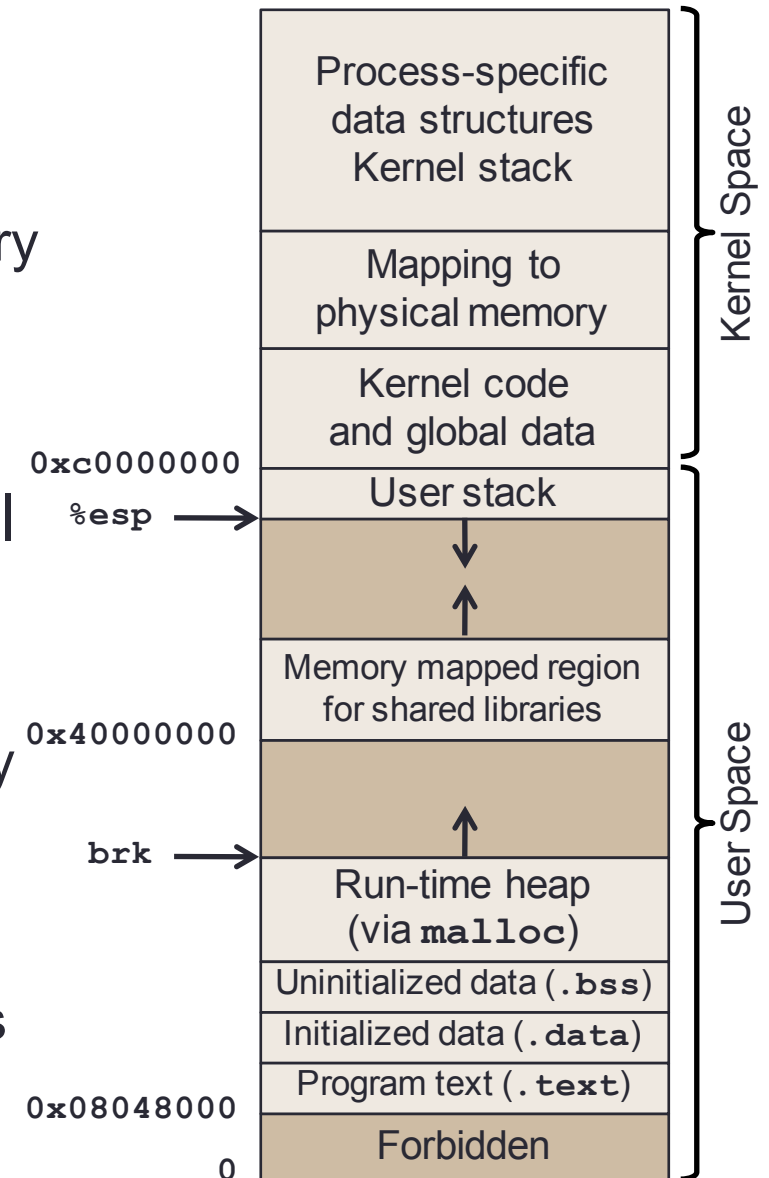
System Calls: Security Holes

- Very important to verify all addresses that come from user-mode programs:
 - Addresses must be in userspace!
 - If an address is in kernel space, it's an access violation
- Fast way to verify addresses:
 - Make sure the address is below the kernel / user address boundary! (e.g. 0xc0000000 in Linux/Pintos)



System Calls and Page Faults

- Addresses below the kernel / user boundary could still be invalid...
 - e.g. pass a pointer to unallocated memory to a `read()` system call
 - e.g. pass a pointer to read-only memory to a `write()` system call
- OS will see a page fault or a general protection fault within the kernel
- Problem: this isn't always an error!
 - Many OSes don't allocate virtual memory pages until they are actually accessed
 - Private copy-on-write pages are marked read-only; first write attempt causes the page to be copied for the writing process



System Calls and Page Faults (2)

- Aside:
 - In the Pintos system-call lab, virtual memory management isn't completed yet, so a page fault does mean an invalid address 😊
- The OS may see memory faults within the kernel:
 - Sometimes these are valid scenarios
 - Sometimes it's an invalid pointer passed to a syscall 😞
 - Sometimes it is a kernel bug 😞 😞
- Assume there is a way to identify valid scenarios...
 - (We will examine that question in a few weeks)
- *How do we distinguish between the remaining two cases?*

System Calls and Page Faults (3)

- How to distinguish between:
 - Faults caused by invalid addresses passed to system calls
 - Faults caused by kernel bugs
- Linux has a very interesting solution to this problem
- *How much kernel code actually interacts with user space?*
 - (Remember, CPU state of user processes is saved onto the kernel stack, which is in kernel space)

System Calls and Page Faults (3)

- The amount of kernel code that interacts with user space is actually very small...
- Linux kernel keeps an **exception table**, which records the addresses of all instructions that touch user space
- In the fault handler, consult the exception table:
 - If the faulting instruction is in the exception table, then the user program passed the kernel a bad pointer
 - Otherwise, it's a kernel bug ☹
- **Aside:** if it's a kernel bug, Linux performs a **kernel oops**
 - Print out suitable info for a kernel developer to debug the error, and log it to the system log
 - Then terminate the process!
 - Keeps kernel bugs from bringing down the entire system...

Example Kernel Oops

```

< Your System ate a SPARC! Gah! >
-----
      ^  ^
      (xx)\
      ( )\
      U  || - - - - w  ||
          ||           ||

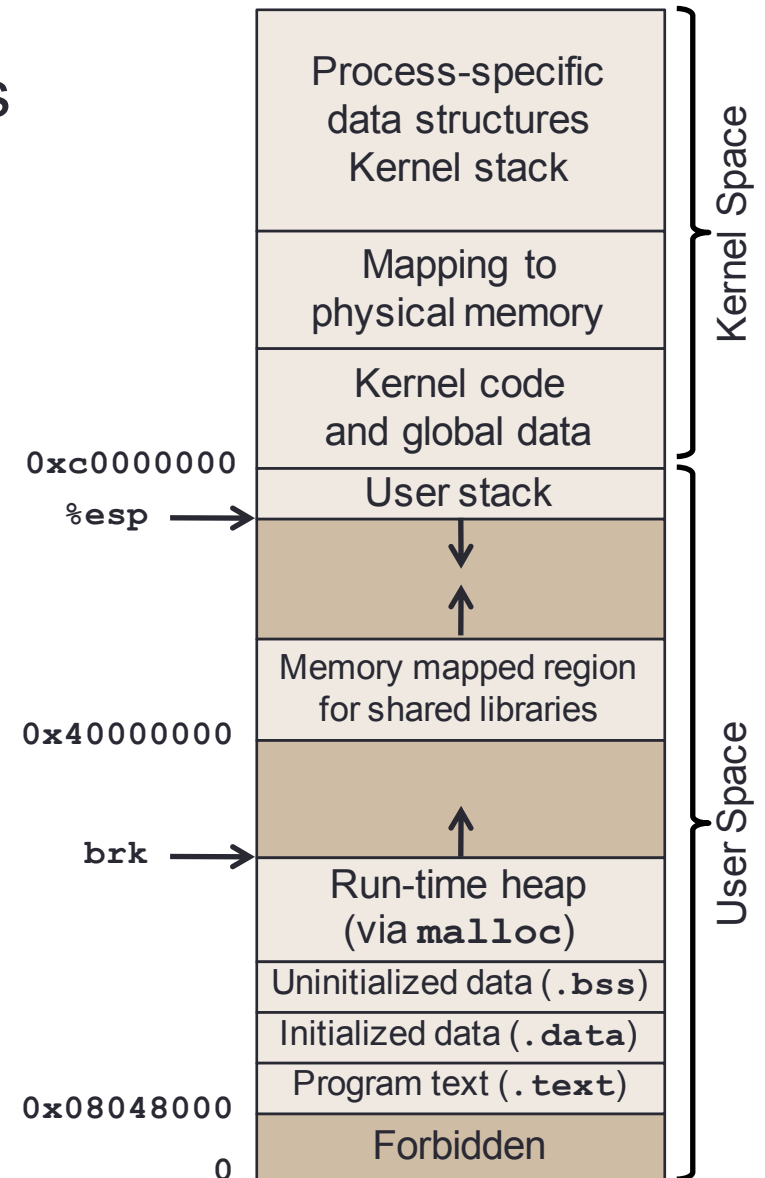
sshd (pid 19569): Protection id trap (code 27)

YZrvWESTHLNXBCVMcbcbcbcb0GFRQPDI
PSW: 00000000000000011011111101000001111 Not tainted
r00-03  00000000 10435a40 101820e4 1d9cfba0
r04-07  00000001 0007d0a8 000000c9 1d9cfba0
r08-11  00000000 0007d0b0 00000001 ffffffff2
r12-15  0006ba00 00068c04 0004d800 0006a404
r16-19  00068c04 0006a404 0006b404 00000001
r20-23  00000001 00000003 0007d0ae 1d9cfbae
r24-27  00000000 00000001 1be099e0 10347010

```

Next Time

- Kernels maintain complex structures on behalf of user processes...
- Kernel has a very limited amount of memory for dynamic allocation
 - Kernel has a fixed memory area for data that pertains to all processes
 - And allocation needs to be fast
- Next time: kernel allocators



Next Week

- Two fun topics next week:
- How signal handling works (1 lecture)
- Kernel allocators: how memory allocations are managed within the kernel (2 lectures)